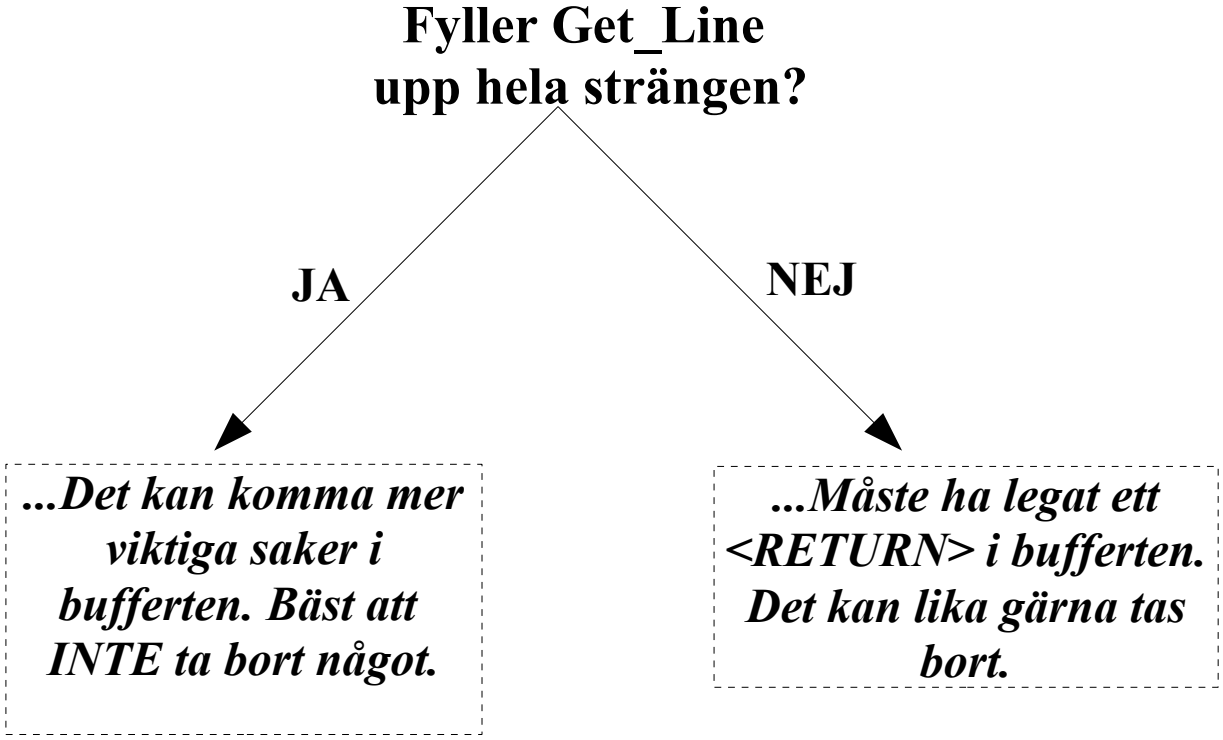
Get_Line

Varför finns Get_Line egentligen?

Hur fungerar Get_Line?

Betrakta vad användaren matar in och ställ frågan:



Alltså:
Get_Line fungerar olika beroende på vad vi matar in!

if-satsen

Vad är skillnaden mellan detta:

```
if x > 10 then
    ...
end if;

if x < -10 then
    ...
end if;

if x = 0 then
    ...
end if;
```

... och detta?

```
if x > 10 then
    ...
elsif x < -10 then
    ...
elsif x = 0 then
    ...
end if;
```

if-satsen

Vad är skillnaden mellan detta:

```
if Villkor1 then
   if Villkor2 then
      Do_Something;
   end if;
end if;
```

...och detta?

```
if Villkor1 and Villkor2 then
   Do_Something;
end if;
```

Kan man få effekten av det första, men samtidigt smidigheten av det senare?

Vad är skillnaden mellan detta:

```
if Villkor1 then
   Do_Something;
elsif Villkor2 then
   Do_Something;
end if;
```

...och detta?

```
if Villkor1 or Villkor2 then
   Do_Something;
end if;
```

Igen: Kan man få det bästa av två världar?

for-satsen

Styrvariabeln **är en variabeln som håller koll på vilket varv man är på i loopen. Variabeln deklareras automatiskt då satsen skrivs men finns bara inne i loopen.**

```
for Styr in 1..10 loop
   ...
end loop;
Put(Styr);        --Ger Kompileringsfel!
```

Skulle det finnas en variabel med samma namn sedan tidigare, kommer denna att *överskuggas* i for-satsen. T.ex:

```
for I in 3..4 loop
   for I in 1..2 loop
      Put(I, Width => 2);
   end loop;
end loop;
```

Det `I` som skrivs ut är det som har deklareras i den inre loopen. Utskriften blir: `1 2 1 2`

exit-satsen

Med exit-satsen kan man avbryta loopar (for, while, loop). När man når exit avbryter man närmsta yttre loop:

```
while Villkor loop
    ...
    exit when Villkor_2;
end loop;
```

```
for I in 1..5 loop
    loop
        ...
        if Villkor then
            exit;--bryter (bara) loop-loopen
        end if;
    end loop;
end loop;
```

Hur gör man om man i en inre loop vill avbryta en yttre / allt ihop? Man använder en *flagga*.

```
   Done : Boolean := False;
begin
   while not Done loop
      loop
         ...
         if Villkor then
            Done := True;
            exit;
         end if;
      end loop;
   end loop;
```

Underprogram

Varför vill vi använda underprogram?

- **Kunna gruppera ihop instruktioner och utföra dem på olika data.**
- **Kunna dela upp ett problem i mindre bitar. Att skapa översikt och *abstraktion* i koden.**
- **Undvika *duplicering* av kodstycken.**
- **Kunna återanvända kod som vi tidigare skrivit. T.ex. genom att skapa ett paket.**
- **Få *modularitet*, lätt kunna byta algoritm t.ex. Bubblesort mot Quicksort.**

Synlighet / Scope

Globala variabler kan ge oönskade effekter...

```
with Ada.Integer_Text_IO;  use ...;

procedure Xyz is
   A, B, C : Integer;
   procedure Swap(A, B : in out Integer) is
   begin
      C := A;
      A := B;
      B := C;
   end Swap;
begin
   A := 1;
   B := 2;
   C := 3;
   Swap(A, B);
   Put(A);
   Put(B);
   Put(C);
end Xyz;
```

Vad kommer ut på skärmen?

2 1 1

Vad gjorde vi för fel?
`C` skall ju vara en lokal variabel i `Swap`.

Globala konstanter kan dock vara bra...

Synlighet / Scope(2)

Globala variabler istället för parametrar?

```
with Ada.Integer_Text_IO;  use …;

procedure Global is

   Global_1 : Integer := 13;
   Global_2 : Integer;

   procedure Set_To_1337 is
   begin
      Global_1 := 1337;
   end Set_To_1337;

begin
   Set_To_1337;
   Put(Global_1);
end Global;
```

`Global_1` ”syns” inne i `set_to_1337`. Därför går ovanstående att kompilera. Vid körning kommer som förväntat talet 1337 ut på skärmen.

Hur gör jag om jag vill anropa `set_to_1337` för `Global_2`? Det går inte, ganska dåligt underprogram...

Synlighet / Scope (3)

Vi lägger till parametrar. Vad gör det då om variablerna ”råkar” synas i underprogrammen?

```
with Ada.Integer_Text_IO;  use …;

procedure Global is

   Global_1 : Integer := 13;

   procedure Set_To_1337(I:in out Integer) is
   begin
      Global_1 := 1337;
   end Set_To_1337;

begin
   Set_To_1337(Global_1);
   Put(Global_1);
end Global;
```

Programmet ovan går att kompilera, verkar oskyldigt. Men vad kommer ut på skärmen? Varför?

Det stora felet är att vi använder den aktuella parametern (`Global_1`) istället för den formella(`I`). Detta borde ge kompileringsfel, men fungerar här eftersom den aktuella parametern är global.

Alltså: Med globala variabler är det lätt att göra fel!

Synlighet / Scope (4)

Det finns fall då detta inte spelar någon roll. Vi byter nu endast ut datatypen Integer mot String.

```
with Ada.Text_IO;    use …;

procedure Global is

   Global_1 : String(1..5) := ”Nisse”;

   procedure Set_To_Elite(I:in out String) is
   begin
      Global_1 := ”Elite”;
   end Set_To_Elite;

begin
   Set_To_Elite(Global_1);
   Put(Global_1);
end Global;
```

Denna gång kommer Elite ut på skärmen!

Anledningen till detta är att (bl.a.) String använder sig av en annorlunda parameteröverföring (ingår ej i kursen).

Verkar det besvärligt att hålla koll på när globala variabler vore okej? Det är det! Undivk dem därför helt och hållet.

Fält

Vi kan se Fält som *Tabeller!*

```
   Week_Day : array(1..7) of String(1..3) :=
        ("Mån", "Tis", "Ons", "Tor",
         "Fre", "Lör", "Sön");
begin
   for I in 1..7 loop
      Put(Week_Day(I));
      Put(' ');
   end loop;
```

Ger utskriften: `Mån Tis Ons Tor Fre Lör Sön`

Det kanske såg ut som om `Week_Day` vore en funktion?

Det är ju för att funktionsanrop och indexering av fält skrivs lika dant (med parenteser). Därför kan en kompilator ibland blanda ihop fält och funktioner i sina felmeddelanden!

I fallet ovan borde `Week_Day` ha varit en konstant. Vill vi använda den i fler funktioner är det då OK at lägga den globalt.

Fält(2)

Fler varianter:

```
Has_Occured : array(0..9) of Boolean :=
     (others => False);
```

0	1	2	3	4	5	6	7	8	9
F	F	F	F	F	F	F	F	F	F

```
Char_Count : array (Character) of Natural :=
     (others => 0);
```

...	'6'	'7'	'8'	'9'	':'	';'	'<'	'='	'>'	'?'	'@'	'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	...
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

```
Ränta : array (2003..2012) of Float :=
    (0.0025, 0.0024, 0.0033, 0.004, 0.0039
    (0.0031, 0.0019, 0.002, 0.0024, 0.002);
```

OBS! Dessa fält har alla *anonym arraytyp*. Det går t.ex inte att skicka dem som parametrar.

Ofta är det lättast att skapa sig en egen typ.

Paket, tips & trix

Hur gör man för att slippa strul med filer? T.ex. att specifikationsfilen (.ads) och body-filen (.adb) inte stämmer överens?

1. **Börja med att skriva allt (typer och underprogram) i ett huvudprogram. Då är det lätt att testa den kod man skrivit.**

2. **När du är nöjd med koden: Klipp ut underprogrammen ur huvudprogrammet och lägg dem i .adb-filen. Lägg eventuella typdefinitioner i .ads-filen.**

3. **Kopiera varje funktions/procedurehuvud i .adb-filen till .ads-filen (lägg dem ovanför ”private”).**

4. **Glöm inte att göra ”with” (och ”use”) på ditt nya paket i huvudprogrammet.**

5. **Då du kompilerar huvudprogrammet, kommer paketet att kompileras automatiskt.**

OBS! Föredra att göra datatypen privat i paketet.

Här hjälper emacs oss igen. Dela emacs-fönstret för att se fler filer samtidigt med C-x 2 och C-x 3.

Ibland behöver ett paket inte en .adb-fil. Närdå?